Date=30/11/2020
Lecture By=Manish Mahant
Subject ⇒Redux Data Flow

IN PREVIOUS LECTURE **(QUICK RECAP) Date-26/11/2020**	**In Today's Lecture (Overview)**
Installation Provider Counter In redux Store In Redux Creating a Store Loading Initial State Dispatching Actions Inside a Redux Store	Background Concepts State Management Immutability Store Dispatch Selectors Redux Application Data Flow#

Background Concepts

Before we dive into some actual code, let's talk about some of the terms and concepts you'll need to know to use Redux.

State Management

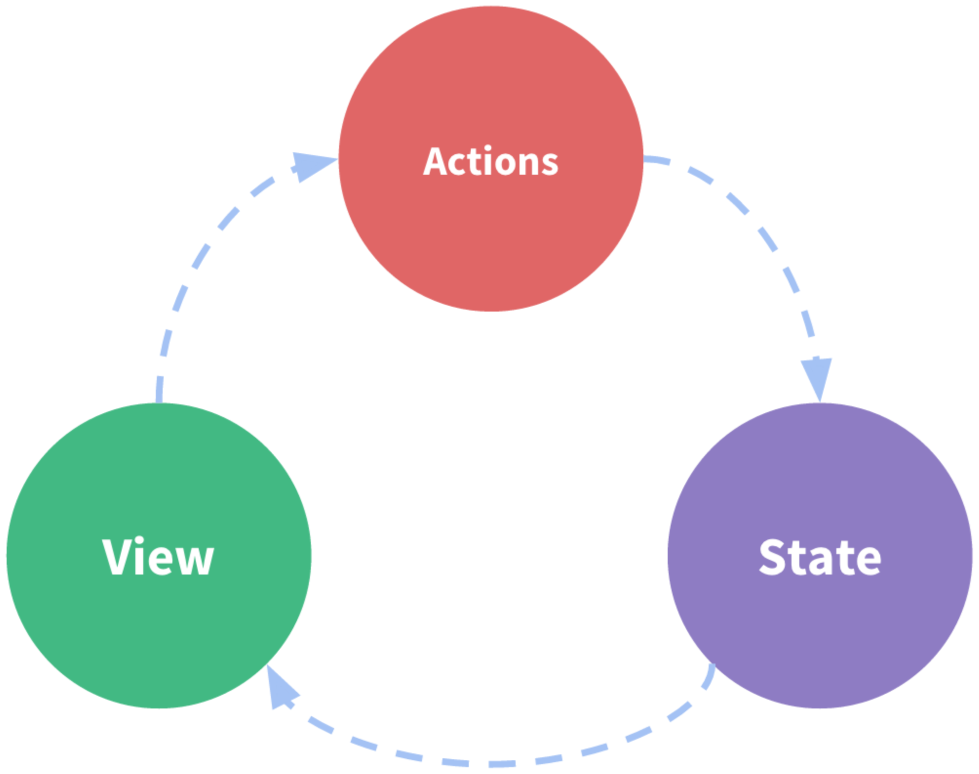
Let's start by looking at a small React counter component. It tracks a number in component state, and increments the number when a button is clicked:

```
function Counter() {
    // State: a counter value
    const [counter, setCounter] = useState(0)

    // Action: code that causes an update to the state when something
happens
    const increment = () => {
      setCounter(prevCounter => prevCounter + 1)
    }

    // View: the UI definition
    return (
      <div>
        Value: {counter} <button onClick={increment}>Increment</button>
      </div>
    )
  }
```
It is a self-contained app with the following parts:

- The state, the source of truth that drives our app;
- The view, a declarative description of the UI based on the current state
- The actions, the events that occur in the app based on user input, and trigger updates in the state

This is a small example of "one-way data flow":

- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state



However, the simplicity can break down when we have multiple components that need to share and use the same state, especially if those components are located in different parts of the application. Sometimes this can be solved by "lifting state up" to parent components, but that doesn't always help.

One way to solve this is to extract the shared state from the components, and put it into a centralized location outside the component tree. With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

By defining and separating the concepts involved in state management and enforcing rules that maintain independence between views and states, we give our code more structure and maintainability.

This is the basic idea behind Redux: a single centralized place to contain the global state in your application, and specific patterns to follow when updating that state to make the code predictable.

Immutability

"Mutable" means "changeable". If something is "immutable", it can never be changed.
JavaScript objects and arrays are all mutable by default. If I create an object, I can change the contents of its fields. If I create an array, I can change the contents as well:

```
const obj = { a: 1, b: 2 }
// still the same object outside, but the contents have changed
obj.b = 3

const arr = ['a', 'b']
// In the same way, we can change the contents of this array
arr.push('c')
arr[1] = 'd'
```

This is called *mutating* the object or array. It's the same object or array reference in memory, but now the contents inside the object have changed.
In order to update values immutably, your code must make *copies* of existing objects/arrays, and then modify the copies.
We can do this by hand using JavaScript's array / object spread operators, as well as array methods that return new copies of the array instead of mutating the original array:

Store#

The current Redux application state lives in an object called the store .
The store is created by passing in a reducer, and has a method called getState that returns the current state value:

```
import { configureStore } from '@reduxjs/toolkit'

const store = configureStore({ reducer: counterReducer })

console.log(store.getState())
// {value: 0}
```

Dispatch

The Redux store has a method called dispatch. The only way to update the state is to call store.dispatch() and pass in an action object. The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value:

Selectors#

Selectors are functions that know how to extract specific pieces of information from a store state value. As an application grows bigger, this can help avoid repeating logic as different parts of the app need to read the same data:

Redux Application Data Flow#

Earlier, we talked about "one-way data flow", which describes this sequence of steps to update the app:

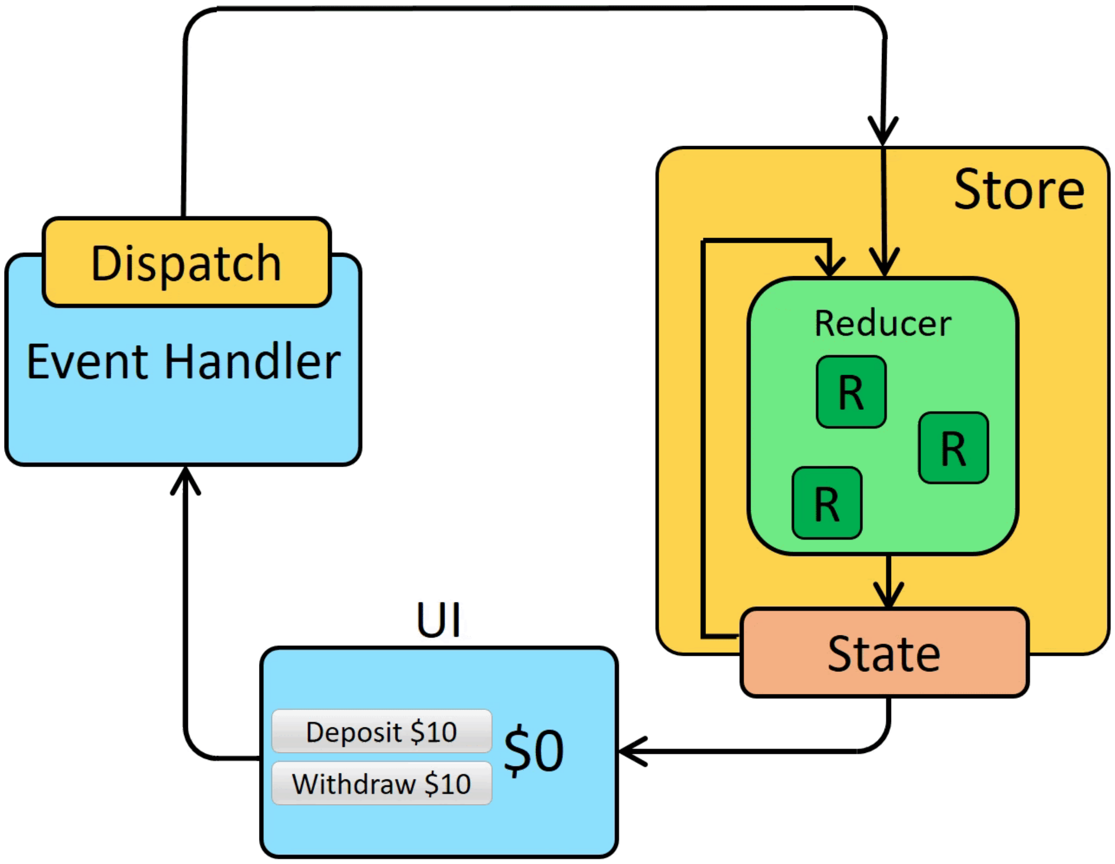
- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state

For Redux specifically, we can break these steps into more detail:

- Initial setup:
    - A Redux store is created using a root reducer function
    - The store calls the root reducer once, and saves the return value as its initial state
    - When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.
- Updates:
    - Something happens in the app, such as a user clicking a button
    - The app code dispatches an action to the Redux store, like dispatch({type: 'counter/incremented'})
    - The store runs the reducer function again with the previous state and the current action, and saves the return value as the new state

    - The store notifies all parts of the UI that are subscribed that the store has been updated
    - Each UI component that needs data from the store checks to see if the parts of the state they need have changed.
    - Each component that sees its data has changed forces a re-render with the new data, so it can update what's shown on the screen

Here's what that data flow looks like visually:


Dispatch
Event Handler
Store
Reducer
R
R
R
State
UI
Deposit $10
Withdraw $10
$0